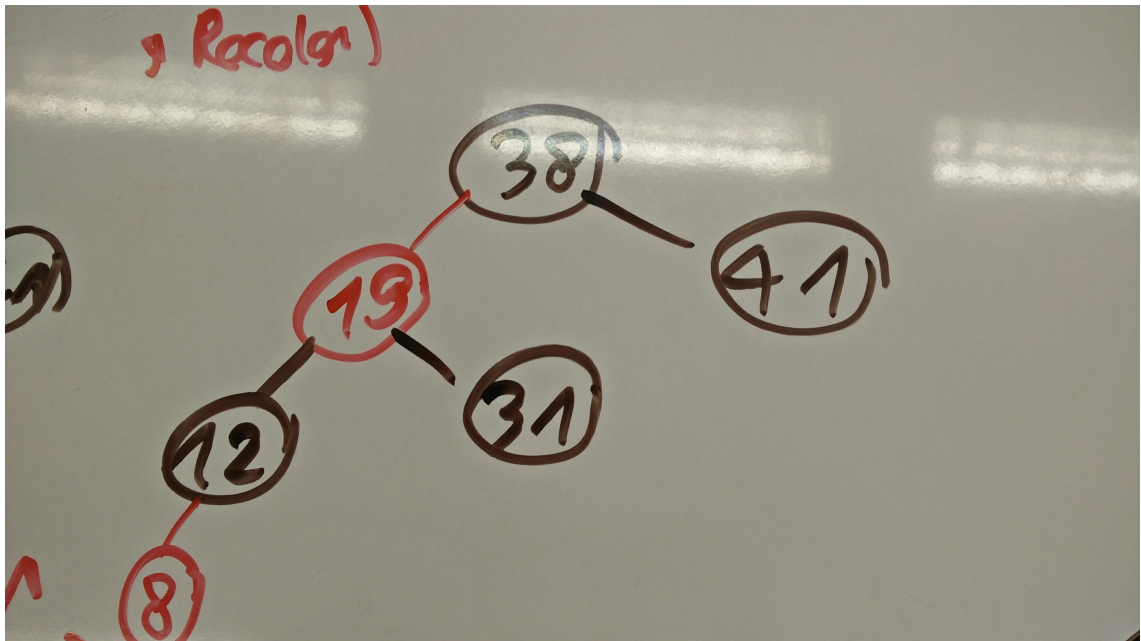


CS 271 - Project 0111

James Le

April 20, 2017

1. Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, and 8 into an initially empty red-black tree.



2. Prove that the longest path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest path from x to a descendant leaf.

Solution: From the red-black properties, we have that every simple path from node x to a descendant leaf has the same number of black nodes and that red nodes do not occur immediately next to each other on such paths. Then the shortest possible simple path from node x to a descendant leaf will have all black nodes, and the longest possible simple path will have alternating black and red nodes. Since the leaf must be black, there are at most the same number of red nodes as black nodes on the path.

3. Implement a red-black tree template class. Your implementation should follow the same guidelines as the binary search tree and include the same public methods.
4. Write another template class implementation of a Dictionary ADT that inherits from your red-black tree template class. Your class should include the same public methods as your Dictionary template classes in the previous projects/
5. Building on the comparison performed in last project, compare the time it takes to insert all of the movies with the red-black tree, hash table, and binary search tree implementations. Explain the results.

Binary Search Tree: 11.529 seconds

The binary search tree is significantly slower than either other data structure because the order of the movies in the movie file creates the worst case scenario tree. This occurs because the movies are in alphabetical order, which is effectively increasing order. Hence, each element inserted is greater than the element inserted before it, so inserting requires the function to

iterate to the right through the entire tree. As constructing the dictionary calls insert multiple times, the function ends up iterating through successively longer trees.

Hash Table: 0.144 seconds

Due to the effective hash function, each movie is assigned a very unique bucket, containing only a small number of additional movies. Since the buckets have almost the same number of movies, the time to go through each list within a bucket is almost the same in each bucket. Since each movie is inserted at the head of its respective bucket, inserting each movie is relatively fast, as the insertion method does not need to iterate to the end of the list.

Red-Black Tree: 1.188 seconds

The red-black tree adjusts itself when new movies are inserted in order to preserve the black height property. This means that iterating to any leaf from the root takes effectively the same amount of time. This self-balancing means that unlike the binary search tree, the red-black tree does not end up as the worst case scenario tree. The tree ends up much more spread out, so there are fewer items to iterate through with each insert. However, the self-balancing adds some extra time to the insertion, so the hash table is quicker.

```
// node.h
// James Le
// Project 0111
// CS 271: Data Structure

#ifndef NODE_H
#define NODE_H

#include <iostream>
#include <string>

template <class KeyType>
class Node
{
public:
    Node();
    Node(KeyType *initKey);
    Node(KeyType *initKey, std::string initColor);
    Node(std::string initColor);

    KeyType *key;
    Node<KeyType> *left;
    Node<KeyType> *right;
    Node<KeyType> *parent;
    std::string color;
};

#include "node.cpp"
#endif
```

```
// node.cpp
// James Le
// Project 0111
// CS 271: Data Structure

#include <string>
#include <iostream>
#include <cstdlib>

using namespace std;

template <class KeyType>
Node<KeyType>::Node()
{
    key = NULL;
    left = NULL;
    right = NULL;
    parent = NULL;
    color = "Red";
}

template <class KeyType>
Node<KeyType>::Node(KeyType *initKey)
{
    key = initKey;
    left = NULL;
    right = NULL;
    parent = NULL;
    color = "Red";
}

template <class KeyType>
Node<KeyType>::Node(KeyType *initKey, string initColor)
{
    key = initKey;
    left = NULL;
    right = NULL;
    parent = NULL;
    color = initColor;
}

template <class KeyType>
Node<KeyType>::Node(string initColor)
{
    key = NULL;
    left = NULL;
    right = NULL;
    parent = NULL;
    color = initColor;
}
```

```
// RBT.h
// James Le
// Project 0111
// CS 271: Data Structure

#ifndef RBT_H
#define RBT_H

#include <iostream>
#include "node.h"
using namespace std;

template <class KeyType>
class RBT
{
public:
    /*-----Constructor and Destructor-----*/
    RBT(); // default constructor
    RBT(const RBT<KeyType>& rbt); // copy constructor
    ~RBT(); // destructor

    /*-----Public Functions-----*/
    bool empty() const; // return true if empty; false otherwise
    KeyType* get(const KeyType& k); // return first element with key equal to k
    void insert(KeyType *k); // insert k into the tree

    KeyType* maximum() ; // return the maximum element
    KeyType* minimum() ; // return the minimum element
    KeyType* successor(const KeyType& k) ; // return the successor of k
    KeyType* predecessor(const KeyType& k) ; // return the predecessor of k

    std::string preOrder() const; // return string of elements from a preorder traversal
    std::string toString() const; // return string of elements from an inorder traversal
    std::string postOrder() const; // return string of elements from a postorder traversal

    RBT<KeyType>& operator=(const RBT<KeyType>& rbt); // assignment operator

private:
    /*-----Class Variables-----*/
    Node<KeyType> *root; // root node
    Node<KeyType> *nil; // nil node

    /*-----Private Functions-----*/
    void deleteNode(Node<KeyType> *node); // delete helper function
    Node<KeyType> *copy(Node<KeyType> *node, Node<KeyType> *p, Node<KeyType> *otherNil); //
    copy helper function
    void ins(KeyType *k, Node<KeyType> *node); // insert helper function

    std::string pre(Node<KeyType> *node) const; // preOrder toString helper function
    std::string order(Node<KeyType> *node) const; // inOrder toString helper function
    std::string post(Node<KeyType> *node) const; // postOrder toString helper function

    void insertFixup(Node<KeyType> * node); // insert fixup to help balance the tree black-
    height-wise
    void leftRotate(Node<KeyType> *node); // left rotation to help rotate the tree to the l
    eft
    void rightRotate(Node<KeyType> *node); // right rotation to help rotate the tree to the
    right

    Node<KeyType> *getNode(const KeyType& k, Node<KeyType> *node); // get helper function
    KeyType *max(Node<KeyType> *node); // maximum method helper function
    KeyType *min(Node<KeyType> *node); // minimum method helper function
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const RBT<KeyType>& rbt); // ostream opera
tor

class Empty { };
class Key { };
```

```
#include "RBT.cpp"
```

```
#endif
```

```
// RBT.cpp
// James Le
// Project 0111
// CS 271: Data Structure

#ifndef RBT_CPP
#define RBT_CPP

#include <string>
#include <sstream>
#include <iostream>
#include <cstdlib>

using namespace std;

/*=====
Constructor
Precondition: None
Postcondition: A constructed RBT with root initialized
to NULL and a nil node to an empty black node.
=====*/
template <class KeyType>
RBT<KeyType>::RBT()
{
    root = NULL;
    nil = new Node<KeyType>("Black");
}

/*=====
Copy Constructor
Precondition: A constructed RBT object
Postcondition: A copy constructor of a new RBT that is
a copy of the RBT passed in as a parameter.
=====*/
template <class KeyType>
RBT<KeyType>::RBT(const RBT<KeyType>& rbt)
{
    root = NULL;
    nil = new Node<KeyType>("Black");
    root = copy(rbt.root, nil, rbt.nil);
}

/*=====
Copy Method
Precondition: A constructed RBT object
Postcondition: A copy helper method that copies a RBT.
The method is then passed into the RBT copy constructor.
=====*/
template <class KeyType>
Node<KeyType>* RBT<KeyType>::copy(Node<KeyType> *node, Node<KeyType> *p, Node<KeyType> *otherNil)
{
    Node<KeyType> *newNode = new Node<KeyType>(node->key);
    if(node == otherNil)
    {
        return nil;
    }

    newNode->color = node->color;
    newNode->parent = p;
    newNode->left = copy(node->left, newNode, otherNil);
    newNode->right = copy(node->right, newNode, otherNil);
    return newNode;
}

/*=====
Destructor
Precondition: A constructed RBT object
Postcondition: Deallocates the space that was allocated
for the RBT.
```

```
=====*/
template <class KeyType>
RBT<KeyType>::~~RBT()
{
    deleteNode(root);
    delete nil;
}

/*=====
Destroy Method
Precondition: A constructed RBT object
Postcondition: A destroy helper method that destroys a
RBT. The method is then passed into the RBT destructor.
=====*/
template <class KeyType>
void RBT<KeyType>::deleteNode(Node<KeyType> *node)
{
    if(node != nil and node != NULL)
    {
        deleteNode(node->left);
        deleteNode(node->right);
        delete node;
    }
}

/*=====
Get Method
Precondition: A constructed RBT object
Postcondition: RBT is unchanged, and a KeyType* is
returned from the tree.
=====*/
template <class KeyType>
KeyType* RBT<KeyType>::get(const KeyType& k)
{
    Node<KeyType> *toReturn = getNode(k, root);
    return toReturn->key;
}

/*=====
Get Helper Function
Precondition: A constructed RBT object
Postcondition: RBT is unchanged, and a a Node<KeyType*>
is returned from the tree.
=====*/
template <class KeyType>
Node<KeyType>* RBT<KeyType>::getNode(const KeyType& k, Node<KeyType> *node)
{
    if(root == NULL)
    {
        throw Empty();
    }
    if(node->key == NULL)
    {
        throw Key();
    }
    if(k == *(node->key))
    {
        return node;
    }
    else if(k < *(node->key))
    {
        return getNode(k, node->left);
    }
    else
    {
        return getNode(k, node->right);
    }
}

/*=====
```


Empty Method

Precondition: A constructed RBT object

Postcondition: RBT is unchanged, and a boolean is returned based on if the RBT is empty or contains an item.

=====*/

```
template <class KeyType>
```

```
bool RBT<KeyType>::empty() const
```

```
{
    if(root == NULL)
    {
        return true;
    }
    return false;
}
```

/*=====

Maximum Method

Precondition: A constructed RBT object

Postcondition: RBT is unchanged, and a pointer of type KeyType is returned. This pointer points to the max value in the RBT

=====*/

```
template <class KeyType>
```

```
KeyType* RBT<KeyType>::maximum()
```

```
{
    KeyType *toReturn = max(root);
    return toReturn;
}
```

/*=====

Maximum Helper Function

Precondition: A constructed RBT object

Postcondition: RBT is unchanged, and a pointer of type KeyType is returned. This pointer points to the max value in the RBT.

=====*/

```
template <class KeyType>
```

```
KeyType* RBT<KeyType>::max(Node<KeyType> *node)
```

```
{
    if(root == NULL)
    {
        throw Empty();
    }
    if(node == NULL)
    {
        return NULL;
    }
    if(node->right == nil)
    {
        return node->key;
    }
    return max(node->right);
}
```

/*=====

Minimum Method

Precondition: A constructed RBT object

Postcondition: RBT is unchanged, and a pointer of type KeyType is returned. This pointer points to the min value in the RBT

=====*/

```
template <class KeyType>
```

```
KeyType* RBT<KeyType>::minimum()
```

```
{
    KeyType *toReturn = min(root);
    return toReturn;
}
```

/*=====

Minimum Helper Function

Precondition: A constructed RBT object

Postcondition: RBT is unchanged, and a pointer of type KeyType is returned. This pointer points to the min value in the RBT.

```

=====*/
template <class KeyType>
KeyType* RBT<KeyType>::min(Node<KeyType> *node)
{
    if(root == NULL)
    {
        throw Empty();
    }
    if(node == NULL)
    {
        return NULL;
    }
    if(node->left == nil)
    {
        return node->key;
    }
    return min(node->left);
}

/*=====
Successor Method
Precondition: A constructed RBT object
Postcondition: Returns the successor to k of type KeyType from RBT.
=====*/
template <class KeyType>
KeyType* RBT<KeyType>::successor(const KeyType& k)
{
    if(root == NULL)
    {
        throw Empty();
    }
    Node<KeyType> *nodeX = getNode(k, root);
    Node<KeyType> *nodeY;
    if(nodeX->right != nil)
    {
        return min(nodeX->right);
    }
    nodeY = nodeX->parent;
    while(nodeY != nil && nodeX == nodeY->right)
    {
        nodeX = nodeY;
        nodeY = nodeY->parent;
    }
    if(nodeY->key == NULL)
    {
        throw Key();
    }
    return nodeY->key;
}

/*=====
Predecessor Method
Precondition: A constructed RBT object
Postcondition: Returns the predecessor to k of type KeyType from RBT.
=====*/
template <class KeyType>
KeyType* RBT<KeyType>::predecessor(const KeyType& k)
{
    if(root == NULL)
    {
        throw Empty();
    }
    Node<KeyType> *nodeX = getNode(k, root);
    Node<KeyType> *nodeY;
    if(nodeX->left != nil)
    {
        return max(nodeX->left);
    }
    nodeY = nodeX->parent;
    while(nodeY != nil && nodeX == nodeY->left)

```

```

    {
        nodeX = nodeY;
        nodeY = nodeY->parent;
    }
    if(nodeY->key == NULL)
    {
        throw Key();
    }
    return nodeY->key;
}

/*=====
Insert Method
Precondition: A constructed RBT object
Postcondition: A RBT object with KeyType *k inserted into the RBT.
=====*/
template <class KeyType>
void RBT<KeyType>::insert(KeyType *k)
{
    ins(k, root);
}

/*=====
Insert Helper Function
Precondition: A constructed RBT object
Postcondition: A RBT object with KeyType *k inserted into the RBT.
=====*/
template <class KeyType>
void RBT<KeyType>::ins(KeyType *k, Node<KeyType> *node)
{
    Node<KeyType> *newNode = new Node<KeyType>(k);

    if(node == NULL)
    {
        root = newNode;
        root->right = nil;
        root->left = nil;
        root->parent = nil;
        insertFixup(root);
    }
    else if(*k <= *(node->key) && node->left == nil)
    {
        node->left = newNode;
        newNode->parent = node;
        newNode->left = nil;
        newNode->right = nil;
        insertFixup(newNode);
    }
    else if(*k > *(node->key) && node->right == nil)
    {
        node->right = newNode;
        newNode->parent = node;
        newNode->left = nil;
        newNode->right = nil;
        insertFixup(newNode);
    }
    else if(*k <= *(node->key))
    {
        ins(k, node->left);
    }
    else
    {
        ins(k, node->right);
    }
}

/*=====
Insert Fixup Method
Precondition: A constructed RBT object
Postcondition: A RBT that abides by the 5 rules of a RBT.

```

```

=====*/
template <class KeyType>
void RBT<KeyType>::insertFixup(Node<KeyType> * node)
{
    Node<KeyType> *tempNode = new Node<KeyType>;
    while(node->parent->color == "Red")
    {
        if(node->parent == node->parent->parent->left)
        {
            tempNode = node->parent->parent->right;
            if(tempNode->color == "Red")
            {
                node->parent->color = "Black";
                tempNode->color = "Black";
                node->parent->parent->color = "Red";
                node = node->parent->parent;
            }
            else
            {
                if(node == node->parent->right)
                {
                    node = node->parent;
                    leftRotate(node);
                }
                node->parent->color = "Black";
                node->parent->parent->color = "Red";
                rightRotate(node->parent->parent);
            }
        }
        else
        {
            tempNode = node->parent->parent->left;
            if(tempNode->color == "Red")
            {
                node->parent->color = "Black";
                tempNode->color = "Black";
                node->parent->parent->color = "Red";
                node = node->parent->parent;
            }
            else
            {
                if(node == node->parent->left)
                {
                    node = node->parent;
                    rightRotate(node);
                }
                node->parent->color = "Black";
                node->parent->parent->color = "Red";
                leftRotate(node->parent->parent);
            }
        }
    }
    root->color = "Black";
}

/*=====
Left Rotation Method
Precondition: A constructed RBT object
Postcondition: The method rotates the tree to the left in order to
maintain balance of the tree in regards of the black height property.
=====*/
template <class KeyType>
void RBT<KeyType>::leftRotate(Node<KeyType> *node)
{
    Node<KeyType> *tempNode = new Node<KeyType>;
    tempNode = node->right;
    node->right = tempNode->left;
    if(tempNode->left != nil)
    {
        tempNode->left->parent = node;
    }
}

```

```

    }
    tempNode->parent = node->parent;

    if(node->parent == nil)
    {
        root = tempNode;
    }
    else if(node == node->parent->left)
    {
        node->parent->left = tempNode;
    }
    else
    {
        node->parent->right = tempNode;
    }
    tempNode->left = node;
    node->parent = tempNode;
}

/*=====
Right Rotation Method
Precondition: A constructed RBT object
Postcondition: The method rotates the tree to the right in order to
maintain balance of the tree in regards of the black height property.
=====*/
template <class KeyType>
void RBT<KeyType>::rightRotate(Node<KeyType> *node)
{
    Node<KeyType> *tempNode = new Node<KeyType>;
    tempNode = node->left;
    node->left = tempNode->right;
    if(tempNode->right != nil)
    {
        tempNode->right->parent = node;
    }
    tempNode->parent = node->parent;

    if(node->parent == nil)
    {
        root = tempNode;
    }
    else if(node == node->parent->right)
    {
        node->parent->right = tempNode;
    }
    else
    {
        node->parent->left = tempNode;
    }
    tempNode->right = node;
    node->parent = tempNode;
}

/*=====
Assignment Operator
Precondition: A constructed RBT object
Postcondition: The assignment operator sets a RBT that we assign to
another tree and makes it equal to the RBT we already had constructed.
=====*/
template <class KeyType>
RBT<KeyType>& RBT<KeyType>::operator=(const RBT<KeyType>& rbt)
{
    root = NULL;
    nil = new Node<KeyType>("Black");

    if(this != &rbt)
    {
        root = copy(rbt.root, nil, rbt.nil);
    }
    return *this;
}

```

```
}

/*=====
PreOrder Method
Precondition: A constructed RBT object
Postcondition: Returns a string of the elements in the RBT after a
preOrder traversal.
=====*/
template <class KeyType>
string RBT<KeyType>::preOrder() const
{
    string toReturn = pre(root);
    return toReturn.substr(0, toReturn.size() - 2);
}

/*=====
PreOrder Helper Function
Precondition: A constructed RBT object
Postcondition: Returns a string of the elements in the RBT after a
preOrder traversal.
=====*/
template <class KeyType>
string RBT<KeyType>::pre(Node<KeyType> *node) const
{
    stringstream s;
    if(node == nil)
    {
        return "";
    } else {
        if(node != nil)
        {
            if(node == root)
            {
                s << *(node->key) << ":" << node->color << ":" << "Root" << ", ";
            } else {
                s << *(node->key) << ":" << node->color << ", ";
            }
        }
        s << pre(node->left);
        s << pre(node->right);
    }
    string returnString = s.str();
    return returnString;
}

/*=====
InOrder Method
Precondition: A constructed RBT object
Postcondition: Returns a string of the elements in the RBT after an
inOrder traversal.
=====*/
template <class KeyType>
string RBT<KeyType>::toString() const
{
    string toReturn = order(root);
    return toReturn.substr(0, toReturn.size() - 2);
}

/*=====
InOrder Helper Function
Precondition: A constructed RBT object
Postcondition: Returns a string of the elements in the RBT after an
inOrder traversal.
=====*/
template <class KeyType>
string RBT<KeyType>::order(Node<KeyType> *node) const
{
    stringstream s;
    if(node == nil)
    {
```

```

    return "";
} else {
    s << order(node->left);
    if(node != nil)
    {
        if(node == root)
        {
            s << *(node->key) << ":" << node->color << ":" << "Root" << ", ";
        } else {
            s << *(node->key) << ":" << node->color << ", ";
        }
    }
    s << order(node->right);
}
string returnString = s.str();
return returnString;
}

```

```

/*=====
PostOrder Method
Precondition: A constructed RBT object
Postcondition: Returns a string of the elements in the RBT after a
postOrder traversal.
=====*/

```

```

template <class KeyType>
string RBT<KeyType>::postOrder() const
{
    string toReturn = post(root);
    return toReturn.substr(0, toReturn.size() - 2);
}

```

```

/*=====
PostOrder Helper Function
Precondition: A constructed RBT object
Postcondition: Returns a string of the elements in the RBT after a
postOrder traversal.
=====*/

```

```

template <class KeyType>
string RBT<KeyType>::post(Node<KeyType> *node) const
{
    stringstream s;
    if(node == nil)
    {
        return "";
    } else {
        s << post(node->left);
        s << post(node->right);
        if(node != nil)
        {
            if(node == root)
            {
                s << *(node->key) << ":" << node->color << ":" << "Root" << ", ";
            } else {
                s << *(node->key) << ":" << node->color << ", ";
            }
        }
    }
    string returnString = s.str();
    return returnString;
}

```

```

/*=====
Ostream Operator
Precondition: A constructed RBT object
Postcondition: Returns a string of the stream of the elements in the
RBT in an inOrder traversal.
=====*/

```

```

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const RBT<KeyType>& rbt)
{

```

```
    stream << rbt.toString();  
    return stream;  
}  
  
#endif
```



```
// test_rbt.cpp
// James Le
// Project 0111
// CS 271: Data Structure

#include <string>
#include <iostream>
#include <cstdlib>
#include "RBT.h"
#include <cassert>

using namespace std;

void test_const()
{
    RBT<int> jamesle;
}

void test_empty()
{
    RBT<int> jamesle;
    assert(jamesle.empty() == 1);

    int *test1 = new int;
    *test1 = 1;

    jamesle.insert(test1);
    assert(jamesle.empty() == 0);
}

void test_copyConst()
{
    RBT<int> jamesle;

    int *test1 = new int;
    *test1 = 11;
    jamesle.insert(test1);

    int *test2 = new int;
    *test1 = 5;
    jamesle.insert(test2);

    int *test3 = new int;
    *test1 = 13;
    jamesle.insert(test3);

    int *test4 = new int;
    *test1 = 23;
    jamesle.insert(test4);

    int *test5 = new int;
    *test1 = 57;
    jamesle.insert(test5);

    int *test6 = new int;
    *test1 = 9;
    jamesle.insert(test6);

    int *test7 = new int;
    *test1 = 19;
    jamesle.insert(test7);

    int *test8 = new int;
    *test1 = 92;
    jamesle.insert(test1);

    assert(jamesle.toString() == "5:Black, 9:Red, 11:Black:Root, 13:Black, 19:Red, 23:Red,
57:Black, 92:Red");

    RBT<int> jamesle2(jamesle);
```

```
    assert(jamesle2.toString() == "5:Black, 9:Red, 11:Black:Root, 13:Black, 19:Red, 23:Red,
57:Black, 92:Red");
}

void test_get()
{
    RBT<int> jamesle;

    try
    {
        throw jamesle.get(11);
    }
    catch(Empty Error)
    {
        cerr << "Error! Trying to get a value in an empty RBT" << endl;
    }

    int *test1 = new int;
    *test1 = 2378;
    jamesle.insert(test1);

    int *test2 = new int;
    *test1 = 5;
    jamesle.insert(test2);

    int *test3 = new int;
    *test1 = 729;
    jamesle.insert(test3);

    int *test4 = new int;
    *test1 = 311;
    jamesle.insert(test4);

    int *test5 = new int;
    *test1 = 5642;
    jamesle.insert(test5);

    int *test6 = new int;
    *test1 = 126;
    jamesle.insert(test6);

    assert(*(jamesle.get(729)) == 729);

    try
    {
        throw jamesle.get(11);
    }
    catch (Key Error)
    {
        cerr << "Error! Trying to get a value that is not in RBT" << endl;
    }
}

void test_insert()
{
    RBT<int> jamesle;

    int *test1 = new int;
    *test1 = 1;
    jamesle.insert(test1);

    int *test2 = new int;
    *test2 = 2;
    jamesle.insert(test2);

    int *test3 = new int;
    *test3 = 3;
    jamesle.insert(test3);

    int *test4 = new int;
```

```
*test4 = 4;
jamesle.insert(test4);

int *test5 = new int;
*test5 = 5;
jamesle.insert(test5);

int *test6 = new int;
*test6 = 6;
jamesle.insert(test6);

int *test7 = new int;
*test7 = 7;
jamesle.insert(test7);

int *test8 = new int;
*test8 = 8;
jamesle.insert(test8);

int *test9 = new int;
*test9 = 9;
jamesle.insert(test9);

int *test10 = new int;
*test10 = 10;
jamesle.insert(test10);

int *test11 = new int;
*test11 = 11;
jamesle.insert(test11);

int *test12 = new int;
*test12 = 12;
jamesle.insert(test12);

int *test13 = new int;
*test13 = 13;
jamesle.insert(test13);

int *test14 = new int;
*test14 = 14;
jamesle.insert(test14);

int *test15 = new int;
*test15 = 15;
jamesle.insert(test15);

int *test16 = new int;
*test16 = 16;
jamesle.insert(test16);

int *test17 = new int;
*test17 = 17;
jamesle.insert(test17);

int *test18 = new int;
*test18 = 18;
jamesle.insert(test18);

assert(jamesle.toString() == "1:Black, 2:Black, 3:Black, 4:Red, 5:Black, 6:Black, 7:Black, 8:Black:Root, 9:Black, 10:Black, 11:Black, 12:Red, 13:Black, 14:Black, 15:Black, 16:Red, 17:Black, 18:Red");
}

void test_oper()
{
    RBT<int> jamesle;
    RBT<int> jamesle2;

    int *test1 = new int;
```

```
*test1 = 4;
jamesle.insert(test1);

int *test2 = new int;
*test2 = 2;
jamesle.insert(test2);

int *test3 = new int;
*test3 = 5;
jamesle.insert(test3);

int *test16 = new int;
*test16 = 16;
jamesle.insert(test16);

int *test17 = new int;
*test17 = 17;
jamesle.insert(test17);

int *test18 = new int;
*test18 = 18;
jamesle.insert(test18);

assert(jamesle.toString() == "2:Black, 4:Black:Root, 5:Black, 16:Red, 17:Black, 18:Red"
);
assert(jamesle.empty() == 1);

jamesle2 = jamesle;
assert(jamesle.toString() == "2:Black, 4:Black:Root, 5:Black, 16:Red, 17:Black, 18:Red"
);
assert(jamesle2.toString() == "2:Black, 4:Black:Root, 5:Black, 16:Red, 17:Black, 18:Red
");

int *test4 = new int;
*test4 = 12;
jamesle.insert(test4);

assert(jamesle.toString() == "2:Black, 4:Black:Root, 5:Black, 12:Red, 16:Red, 17:Black,
18:Red");
assert(jamesle2.toString() == "2:Black, 4:Black:Root, 5:Black, 16:Red, 17:Black, 18:Red
");
}

void test_max()
{
    RBT<int> jamesle;

    try
    {
        throw jamesle.maximum();
    }
    catch(Empty Error)
    {
        cerr << "Error! Trying to get maximum value in an empty RBT" << endl;
    }

    int *test1 = new int;
    *test1 = 4;
    jamesle.insert(test1);

    int *test2 = new int;
    *test2 = 2;
    jamesle.insert(test2);

    int *test3 = new int;
    *test3 = 5;
    jamesle.insert(test3);

    int *test16 = new int;
    *test16 = 16;
```

```
jamesle.insert(test16);

assert(*(jamesle.maximum()) == 16);
}

void test_min()
{
    RBT<int> jamesle;

    try
    {
        throw jamesle.minimum();
    }
    catch(Empty Error)
    {
        cerr << "Error! Trying to get minimum value in an empty RBT" << endl;
    }

    int *test1 = new int;
    *test1 = 4;
    jamesle.insert(test1);

    int *test2 = new int;
    *test2 = 2;
    jamesle.insert(test2);

    int *test3 = new int;
    *test3 = 5;
    jamesle.insert(test3);

    int *test16 = new int;
    *test16 = 16;
    jamesle.insert(test16);

    assert(*(jamesle.minimum()) == 2);
}

void test_pred()
{
    RBT<int> jamesle;

    try
    {
        throw *jamesle.predecessor(4);
    }
    catch(Empty Error)
    {
        cerr << "The RBT is currently empty!" << endl;
    }

    int *test1 = new int;
    *test1 = 4;
    jamesle.insert(test1);

    int *test2 = new int;
    *test2 = 2;
    jamesle.insert(test2);

    int *test3 = new int;
    *test3 = 5;
    jamesle.insert(test3);

    int *test16 = new int;
    *test16 = 16;
    jamesle.insert(test16);

    try
    {
        throw *jamesle.predecessor(2);
    }
}
```

```
    catch(Key Error)
    {
        cerr << "There is no predecessor for this value!" << endl;
    }

    try
    {
        throw *jamesle.predecessor(7);
    }
    catch(Key Error)
    {
        cerr << "Value inserted not in RBT!" << endl;
    }

    assert(*(jamesle.predecessor(4)) == 2);
    assert(*(jamesle.predecessor(5)) == 4);
    assert(*(jamesle.predecessor(16)) == 5);
}

void test_succes()
{
    RBT<int> jamesle;

    try
    {
        throw *jamesle.predecessor(4);
    }
    catch(Empty Error)
    {
        cerr << "The RBT is currently empty!" << endl;
    }

    int *test1 = new int;
    *test1 = 4;
    jamesle.insert(test1);

    int *test2 = new int;
    *test2 = 2;
    jamesle.insert(test2);

    int *test3 = new int;
    *test3 = 5;
    jamesle.insert(test3);

    int *test16 = new int;
    *test16 = 16;
    jamesle.insert(test16);

    try
    {
        throw *jamesle.successor(16);
    }
    catch(Key Error)
    {
        cerr << "There is no successor for this value!" << endl;
    }

    try
    {
        throw *jamesle.successor(7);
    }
    catch(Key Error)
    {
        cerr << "Value inserted not in RBT!" << endl;
    }

    assert(*(jamesle.successor(2)) == 4);
    assert(*(jamesle.successor(4)) == 5);
    assert(*(jamesle.successor(5)) == 16);
}
```

```
void test_pre()
{
    RBT<int> jamesle;

    int *test1 = new int;
    *test1 = 1;
    jamesle.insert(test1);

    int *test2 = new int;
    *test2 = 2;
    jamesle.insert(test2);

    int *test3 = new int;
    *test3 = 3;
    jamesle.insert(test3);

    int *test4 = new int;
    *test4 = 4;
    jamesle.insert(test4);

    int *test5 = new int;
    *test5 = 5;
    jamesle.insert(test5);

    int *test6 = new int;
    *test6 = 6;
    jamesle.insert(test6);

    int *test7 = new int;
    *test7 = 7;
    jamesle.insert(test7);

    int *test8 = new int;
    *test8 = 8;
    jamesle.insert(test8);

    assert(jamesle.preOrder() == "4:Black:Root, 2:Red, 1:Black, 3:Black, 6:Red, 5:Black, 7:
Black, 8:Red");
}

void test_toString()
{
    RBT<int> jamesle;

    int *test1 = new int;
    *test1 = 1;
    jamesle.insert(test1);

    int *test2 = new int;
    *test2 = 2;
    jamesle.insert(test2);

    int *test3 = new int;
    *test3 = 3;
    jamesle.insert(test3);

    int *test4 = new int;
    *test4 = 4;
    jamesle.insert(test4);

    int *test5 = new int;
    *test5 = 5;
    jamesle.insert(test5);

    int *test6 = new int;
    *test6 = 6;
    jamesle.insert(test6);

    int *test7 = new int;
```

```
*test7 = 7;
jamesle.insert(test7);

int *test8 = new int;
*test8 = 8;
jamesle.insert(test8);

assert(jamesle.toString() == "1:Black, 2:Red, 3:Black, 4:Black:Root, 5:Black, 6:Red, 7:
Black, 8:Red");
}

void test_post()
{
    RBT<int> jamesle;

    int *test1 = new int;
    *test1 = 1;
    jamesle.insert(test1);

    int *test2 = new int;
    *test2 = 2;
    jamesle.insert(test2);

    int *test3 = new int;
    *test3 = 3;
    jamesle.insert(test3);

    int *test4 = new int;
    *test4 = 4;
    jamesle.insert(test4);

    int *test5 = new int;
    *test5 = 5;
    jamesle.insert(test5);

    int *test6 = new int;
    *test6 = 6;
    jamesle.insert(test6);

    int *test7 = new int;
    *test7 = 7;
    jamesle.insert(test7);

    int *test8 = new int;
    *test8 = 8;
    jamesle.insert(test8);

    assert(jamesle.postOrder() == "1:Black, 3:Black, 2:Red, 5:Black, 8:Red, 7:Black, 6:Red,
4:Black:Root");
}

int main()
{
    test_const();
    test_copyConst();
    test_empty();
    test_get();
    test_insert();
    test_oper();
    test_max();
    test_min();
    test_pred();
    test_succes();
    test_pre();
    test_toString();
    test_post();
    return 0;
}
```



```
// dict.h
// James Le
// Project 0111
// CS 271: Data Structure

#ifndef DICTIONARY
#define DICTIONARY

#include <iostream>
#include "RBT.h"

template <class KeyType>
class Dictionary : public RBT<KeyType>
{
public:
    Dictionary() : RBT<KeyType> () { }; // constructor

    using RBT<KeyType>::insert;
    using RBT<KeyType>::get;
    using RBT<KeyType>::toString;
    using RBT<KeyType>::empty;
};

#endif
```

```
// movie.h
// James Le
// Project 0111
// CS 271: Data Structure

#ifndef MOVIE
#define MOVIE

#include "dict.h"
#include <iostream>
#include <string>
#include <stdlib.h>
#include <sstream>
#include <iostream>

class Movie
{
public:
    string title; // string of movie titles
    string cast; // string of cast members

    bool operator==(const Movie& mov) const;
    bool operator<=(const Movie& mov) const;
    bool operator>(const Movie& mov) const;

    std::string toString() const; // toString method
};

std::ostream& operator<<(std::ostream& stream, const Movie& movie); // ostream operator

#endif
```

```
// movie.cpp
// James Le
// Project 0111
// CS 271: Data Structure

#include "movie.h"
#include "math.h"

using namespace std;

bool Movie::operator==(const Movie& mov) const
{
    if(title == mov.title)
    {
        return true;
    }
    return false;
}

bool Movie::operator<=(const Movie& mov) const
{
    if(this->title <= mov.title)
    {
        return true;
    } else {
        return false;
    }
}

bool Movie::operator>(const Movie& mov) const
{
    if(this->title > mov.title)
    {
        return true;
    } else {
        return false;
    }
}

string Movie::toString() const
{
    stringstream s;

    s << title << ": ";
    s << cast << '\n';

    string returnString = s.str();
    return returnString.substr(0, returnString.size() - 2);
}

ostream& operator<<(ostream& stream, const Movie& movie)
{
    stream << movie.toString() << "\n";
    return stream;
}
```

```
// query_movies.cpp
// James Le
// Project 0111
// CS 271: Data Structure

#include <iostream>
#include <fstream>
#include <string>
#include <stdlib.h>
#include <sstream>
#include <sys/time.h>
#include "movie.cpp"

using namespace std;

int main()
{
    ifstream infile; // file I am reading from
    infile.open("movies_mpaa.txt"); // name of file I am reading from
    Dictionary<Movie> movieTable; // create inf dictionary that holds movie names in slots

    string line;

    timeval timeBefore, timeAfter; // timeval type defined in sys/time.h
    long diffSeconds, diffUSEconds; // elapsed seconds and microseconds
    gettimeofday(&timeBefore, NULL); // get the time before

    while(getline(infile, line)) // gets each line
    {
        Movie *movie = new Movie;
        int count = 0;
        while(line[count] != '\t')
        {
            count++;
        }
        movie->title = line.substr(0, count); // create title string from letter 0 up to length of title
        movie->cast = line.substr(count + 1); // create cast string from one letter after the title until the end of the line
        movieTable.insert(movie); // inserting the movie object in the dictionary
    }
    infile.close(); // close input file

    gettimeofday(&timeAfter, NULL); // get the time after
    diffSeconds = timeAfter.tv_sec; - timeBefore.tv_sec; // elapsed seconds
    diffUSEconds = timeAfter.tv_usec; - timeBefore.tv_usec; // elapsed microseconds
    double time = diffSeconds + diffUSEconds / 100000.0; // total elapsed time
    cout << time << endl;

    string movieTitle;
    cout << "Enter a movie title: ";
    getline(cin, movieTitle); // prompting user to enter movie title
    while(true)
    {
        if(movieTitle == "Quit")
        {
            break;
        } else {
            Movie find; // creating a movie object
            find.title = movieTitle;
            Movie *print = movieTable.get(find); // find user prompted movie title

            cout << endl << "Cast of the movie: " << movieTitle << endl << endl;

            cout << print->cast << endl; // print cast of the movie
            cout << endl;
            cout << "Enter another movie title you want to find or type Quit: ";
            getline(cin, movieTitle); // prompting user to enter another movie title
        }
    }
}
```

```
    return 0;  
}
```